MILyBots: Design and Development of Swarm-Robots

Luis Vega
Devin Hughes
Machine Intelligence Lab
325 MAE Building B
PO BOX 116300
Gainesville, FL 32611
(352) 392-6605
luisvegas@ufl.edu
dkhughes@ufl.edu

Camilo Buscaron
Machine Intelligence Lab
325 MAE Building B
PO BOX 116300
Gainesville, FL 32611
(352) 392-6605
cbuscaron@ufl.edu

Dr. Eric M. Schwartz
Dr. A. Antonio Arroyo
Machine Intelligence Lab
PO BOX 116300,
Gainesville, FL 32611
(352) 392-6605
ems@mil.ufl.edu
arroyo@mil.ufl.edu

ABSTRACT

This paper describes the design, development and implementation procedures of a swarm-robotics project at the Machine Intelligence Laboratory (MIL) at the University of Florida. The main objective of this work is to develop a multipurpose and powerful platform for the study and improvement of swarm robotics techniques. The first objective is to produce a set of at least eight small expandable (and easily replicated) robots with on-robot sensory and processing abilities, and with a communication system for off-robot sensors and processing. The second goal is the creation of a cross language platform composed of code written in C, C++ and C#; with a well designed object-oriented platform that closely follows the main paradigms and ideas of object-oriented programming techniques. This paper presents the current state of the ongoing project to create a low-cost, reliable, robust, reusable, movable, size-efficient, power-saving, wireless-capable, and dynamically programmable multi-use research project.

Keywords

MILyBots, Agent, Behavior, Swarm intelligence.

1. INTRODUCTION

The core scientific objective of this project is the development and implementation of autonomous robots that can serve as a novel platform for advance swarm intelligence techniques. Swarm intelligence is an emergent field in robotics that concentrates on the study of groups of robots interacting and accomplishing tasks together. The main difference between traditional robot behavior and a robot swarm's behavior is that no one centralized control dictates the behavior of swarm agents; rather, agents develop a global behavior through interaction with their environment and their peers to accomplish a given task by optimal means [1].

MILyBots is comprised of a set of eight agents that can perform group level tasks while each agent concurrently possesses autonomous movement and control. A wireless communication medium provides data transmission between the agents and a computer. The coordinating computer acquires visual data from the given environment and processes an optimal solution for the agent's given task, thus communicating the acquired data and the behavioral strategy to be taken with each of the robots.

Different activities and testing command simulations are being develop to test the agent's abilities and adaptability to achieve a goal within a series of given boundaries. The implementation of swarm principles, techniques, and the autonomous behavioral achievements of such principles and techniques are the key characteristics that set MILyBots apart from other autonomous robotic research. In this paper we report on how MILyBots were created to satisfy these tasks as well as how each individual underlying part of this project fits in the design and implementation of this project as a whole [2].

2. PROJECT OBJECTIVES

The ultimate goal of the MILyBot project is to create a set of **autonomous agents performing coactive and/or individual goal oriented tasks.** To fully comprehend the meaning of this objective, we first need to understand and achieve all the required objectives that are prerequisites for this final objective. These sub-objectives are described below.

2.1 Implement a robust, reliable and reusable platform

The first goal is the design and implementation of a low cost, small, easy to build, and powerful platform that will facilitate swarm behavior among agents.

2.2 Design and build a powerful and size efficient electrical board

The main design goal on the electrical system's printed circuit board (PCB) is to support all the functionality required by the project while maintaining the required small size of the robots. The PCB design supports 3 analog inputs, 1 serial port, a wireless module, 8 digital I/O ports, a 7-segment LED, 4 PWM generators, and a JTAG programming header.

2.3 Provide a wireless communication mechanism between agents

A mechanism must be provided for the agents to communicate among with each other through wireless messages. This will allow flexibility and abstraction of computing power to an external computer machine and allows the agents to focus only on executing behaviors rather than computing locally what to do next. This promotes loose coupling behaviors and dynamic programming.

2.4 Provide visual recognition algorithms for the agent location, identity and facing direction

Algorithms to analyze pictures taken from a top view camera to an area were the agents are present are necessary. Such technique provides real-time tracking of robot location, identity, and facing direction.

2.5 Create a choreography compiler for dynamic robot behavior

The creation of a VPL (Visual Programming Language) in where the user can dynamically create choreographies for the agents to perform in a synchronized manner is also a necessary component of the system. The VPL will provide an easy way to create choreographies, while dynamic testing the current state and finally permits to compile and run this code, hiding the implementation and details of the compiler.

2.6 Remotely control the agents using a Xbox360 Controller

Non-computer control of agents must be possible. An Xbox360 remote control can provide the users with the ability to dynamically control multiple and different agents. Human controlled robots will offer a new interaction between driven agents and autonomous agents, to provide a further challenge to artificial intelligence of the agents [3].

2.7 Autonomous navigation from visual recognition feedback and wireless communication

Each MILyBot will be provided limited visual information wirelessly from top-view camera for on-board processing. Each MILyBot will use this information, along with other on-robot sensors, to navigate around the MILyBots world, trying to determine her relative location, to obtain her direction of motion, and to ascertain her next movement [4].

2.8 Autonomous surrounding awareness from visual feedback and wireless communication

By querying the visual algorithms, a MILyBot will find the position of obstacles and other agents, with respect to its position, and will autonomously determine her next action [4].

2.9 Autonomous artificial intelligence tasks executions system

An agent must be able to complete a given task. The task is described as two sets, one set of rules and one of goals. Each rule can tell the robot to match location with objects or to behave in a certain manner. The reaching of a goal will signal the completion of the task. By integrating the visual algorithms and a task, the system can interact with the robot autonomously and tell her when the goal is achieved and also provide other feedback that could help the agent to complete the task [6].

2.10 Autonomous agents performing coactive or individual goal oriented tasks.

The final desired objective is to create complex tasks where team interaction is required along with the concept of competing teams [3]. Every other agent (player) in the game can either be a friend or a rival. By combining all the previous goals, the agents will be able to complete complicated tasks that we know as games or sports (such as soccer). The agents will determine how to play each of these games and to act individually or in cooperation [7].

3. MECHANICAL PLATFORM

A crucial part of the MILyBots project is the effective design and development of the physical robots as seen in Figures 1 and 2. The final design shown here underwent a great deal of analysis, testing, and trial and error to determine the best parts and materials.

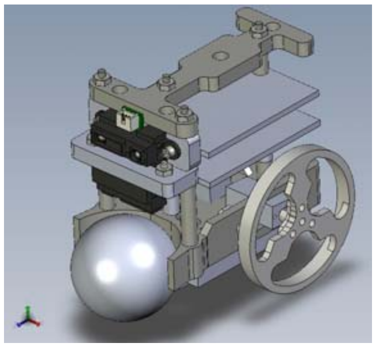


Figure 1 Computer aided designed model

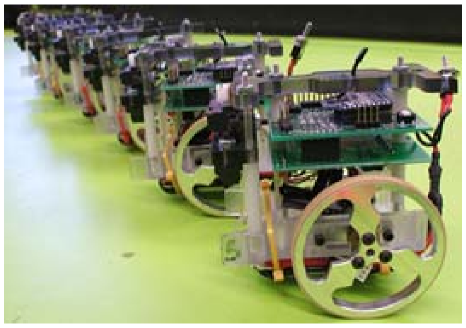


Figure 2 Team of MILyBots

3.1 Main Chassis

The chassis of each MILyBot (see Figure 3) is made of a 2-in wide by 3.5-in long CNC machined polycarbonate frame. This frame serves as a building block for the entire structure of the robot. The chassis was machined to also support the motors, gearboxes, and strategically accommodate the battery pack with all the wiring in an elegant manner. Four 3-in all-threaded rods, arranged in a triangular formation and screwed into the chassis, serve as the supporting columns for the rest of the assembly. A frontal circular opening was made as a pocket to receive balls; this is a crucial physical aspect needed for the development of future research in game playing (e.g., soccer). Polycarbonate was chosen as material due to its low cost and the relative ease of machining.

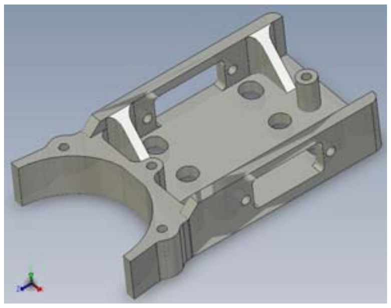


Figure 3 Rendering of the robot's chassis

3.2 Wheels

Each MILyBot rotates and translates by means of two motorized aluminum wheels or radius of 1.25-in. The large wheel size with respect to the entire platform was chosen to achieve efficient rotation. Each wheel is designed in a three spoke arrangement in order to minimize weight without compromising structural strength.

4. ELECTRONIC HARDWARE

The electrical system consists of two custom made compact printed circuitry boards (see Figures 4 and 5) that feature all the electronic system including a microprocessor and a wireless chip. In addition, various light emitting diodes (LEDs) as well as a 7-segment LED are strategically mounted on the boards for debugging purposes.

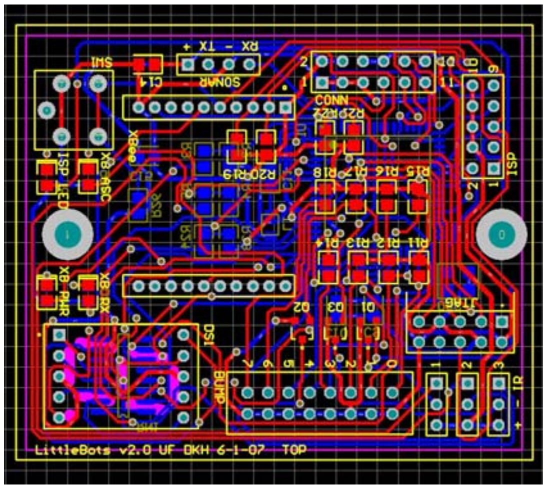


Figure 4 Custom made circuitry board (Top piece)

4.1 Power Supply

Each MILyBot features a 2-cell, 7.4V Lithium Polymer 1250mAh battery pack. Lithium polymer chemistry batteries are preferable over other battery chemistries because of their higher energy density and lower cell count [3]. In order to estimate run time of each robot, a worst case scenario was tested in which a robot was run at full motor speed, under full load, with every electronic component running; under these conditions a single robot last about 2 hours. However, in a typical application, the robots can be expected to have adequate power for 2 ½ to 3 hours.

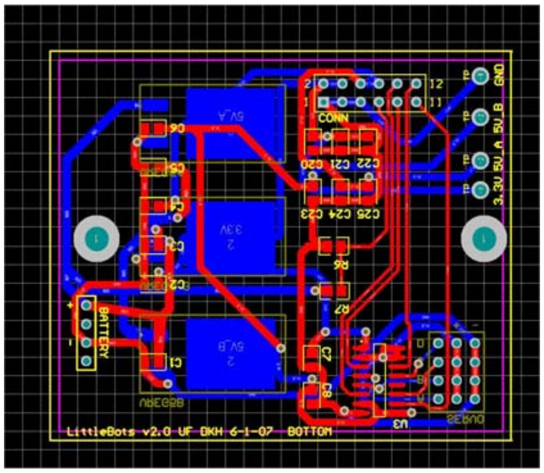


Figure 5 Custom made circuitry board (Bottom piece)

4.2 Microcontroller

Each MILyBot's top printed circuit board has an 8-bit ATmega 128 16MHz processor on the bottom side of the board. This microcontroller processes of all the inputs and outputs, including creating the PWM motor driving signals. This ATmega128 was chosen because of its capabilities that lend themselves to easy implementation of sensor and control units. A JTAG programming port allows the hard-wire programming of the processor.

4.3 Wireless Communication

The wireless communication between the central processing unit and each robot is facilitated by a MaxStream wireless development kit connected to the central processing unit which communicates to each onboard XBee OEM RF Module (Figure 6). The XBee boards are mounted to each MILyBot's main (top) PCB. Information on the packing and unpacking of data will be explained in section 5.2.2, along how it relates to the software aspects of this project.

Figure 6 XBee wireless module

4.4 Motors

Each robot is composed of two Solarbotics GM11 a 56:1 Mini Metal Gear Motor (Figure 7). Low power consumption and excellent torque output make these motors extremely efficient for the purposesof our design.

Figure 7 Solarbotics motor and gearbox assembly

4.5 Sensors

An array of two Sharp GP2D12 infrared sensors (IR) is used to analyze proximity to objects. One sensor faces the front end of the robot and is use for obstacle avoidance behavior. A second IR faces down and is situated just on the top of the ball opening of the chassis to detect when a ball is within the robot's front ring.

4.5.1 Sensor expandability

Future development was taken into consideration with design of the circuitry boards. Additional ports have been provided on the PCBs to allow for the following additional sensors to be added in the future: one sonic range finder, eight bump sensors, and addition additional infrared sensor.

5. SOFTWARE SYSTEM

5.1 Software Architecture

This project requires the design and implementation of three different software systems, each using different programming languages, and a main system that interfaces them together. The first software system represents the behaviors and parameters that are programmed into the robot's microprocessor. This system represents the low-level software system of the project, is coded in C, and targets the ATmega128 microprocessor's architecture.

The second software system provides all the computer vision algorithms required. The vision functions are written using OpenCV, Intel's open source computer vision library (http://intel.com/technology/computing/opencv/), and is written in C++. The design goal for this system is to provide a set of functions that optimize performance and perform each of the desired tasks by using single function calls. This complex function intentionally hides the implementation and details of its inner workings. The code is then compiled and converted into a DLL's (Microsoft's dynamic-link library) so it can be imported from other applications.

The final and most crucial software system is the one in charge of integrating all these systems, and synchronizing them to work together to create and pass messages between them. The final system design goals are to provide a simple graphical user interface for a complex and well designed object-oriented system. This software system, written in C#, is responsible for handling all the artificial intelligence commands and behaviors. The project uses and integrates a multi-language platform software system, each one with independent goals, but targeted to work together. We can refer to the three systems as low-level system (C), mid-level system (C++) and high-level system (C#).

5.2 Low-Level System (LLS)

The main goal of this system is to give the robots the ability to perform different behaviors to allow interaction with the hardware (i.e., motors, LED's, IR's, etc.). The LLS is designed with the goal of encapsulating each action into simple function calls, each of which will cause one and only one result in hardware. For instance, there will be functions that control only one wheel. No function shall control both wheels or assume that they work together. The result of the LLS is a collection of functions that give us individual access to each hardware component. Each function is independent to each other, that means that parallel processing would not affect the functionality of the system. There are two main functions that the LLS provides: full access to all the hardware components and a well defined protocol for wireless communications. The hardware components that need to be controlled are as follows.

- Two Motor Controllers: These motor controllers provide independent movement to the two wheels. Each wheel receives a value between -100 to 100, corresponding to full speed counterclockwise rotation to full speed rotation clockwise, respectively).
- 7-segment LED: This allows the 7-segment LED state to be changed at any time. This is good for debugging or even to display additional information about the robot, e.g., its identity or it running behavior number.
- IR's Sensors: These sensors (up to three) give an analog value proportional to the proximity of an object to the sensor. The ability to query this sensor data is very important in order to provide reliable navigation and obstacle avoidance.
- Digital I/0 ports: The ability to set for read or write up to eight distinct pins on the I/O port headers permits additional hardware components to be added to the system. Some possible uses of these pins are for bumpers (as inputs) or for LEDs (as outputs).
- Serial Communication: The ability to communicate with another serial device is available. The exchange of data can be used for further intelligent behavior. Some possible uses of the serial port are for a sonar ranger finder and a camera.

In addition to the hardware control provided above, the second goal is to create a wireless protocol to allow the robot to receive and send messages from the outside world. A function has been created that handles the incoming packet from the wireless receiver and transform its data into one of the commands that are specified above. By integrating these actions, we can control the behavior of the robot by simply sending the packets with the right data. In order for this to work properly, we need to specify a consistent data packet that will tell the robot an appropriate series of actions.

5.2.1 Data Packet Protocol

The data packet syntax is as follows:

<Command><Parameter><Semicolon>

The <Command> is a two character pair that represents what type of function is commanded. With this data the compiler will look up and then execute the function for which the first two letters match, e.g., MA → motor Command A, MB → motor Command B, QA,QB,QC → Query IR X value , LE → LED set, RX → Receive from serial, SX → Send serially.

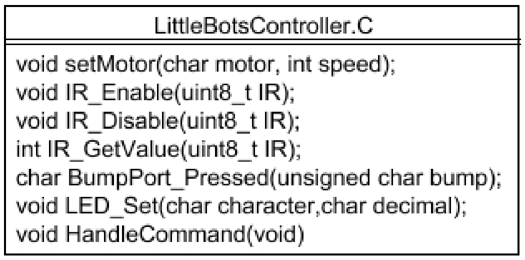
The <Parameter> represents a numerical value that will be passed as the parameter of the calling function determined by the <Command>. Finally a semicolon marks the end of the command. If the command is supposed to return data it will return it as a result of the function call; if not, it will carry on the operation denoted by the <Command> and will execute the specified algorithm to interact with the hardware.

5.2.2 Wireless capabilities:

The agent's Xbee wireless module allows the robot to talk to the coordinator XBee module, which uses the C# code in the high-level system. The Xbee modules allow a two-way communication from the sender to a base board plugged into the computer. The importance of wireless communication is critical since it allows total control of the robot movement and data from an outside computer, which of course can have orders of magnitude of higher capabilities than the embedded microcontrollers on the MILyBots. The MILyBot LLS systems can all the functions necessary to build a packet and to send it to its destination [8].

5.2.3 Design

Figure 8 shows the controller functions, i.e., the UML (Unified Modeling Language) diagram for this component. The heart of all functions is the *HandleCommand* function that continuously runs from the main method. When a message is received through the wireless chip, it checks the proper <Command> in order to perform the desired action. The functions are not generally intended to be visible externally; instead they are written with the purpose of being called by the *HandleCommand()* function when a message is received and will follow the packet rules. Nevertheless, each robot can be used in a standalone application in which all the functions could be used by the programmer in order to hard code a behavior for the robots. (e.g,, obstacle avoidance, end-of-the-world [edge of table] detection). The real purpose behind these functions is to provide a simple interface for a user or developer so that she can program the robots without regard to how each function works. We intend for the robots to run the *HandleCommand()* function indefinitely and act upon received commands from the Xbee module. As far as this system is concerned, it does not care what or how the other systems work, i.e., it works independently from the other systems).



```
LittleBotsController.C
void setMotor(char motor, int speed);
void IR_Enable(uint8_t IR);
void IR_Disable(uint8_t IR);
int IR_GetValue(uint8_t IR);
char BumpPort_Pressed(unsigned char bump);
void LED_Set(char character,char decimal);
void HandleCommand(void)
```

Figure 8 Controller Functions

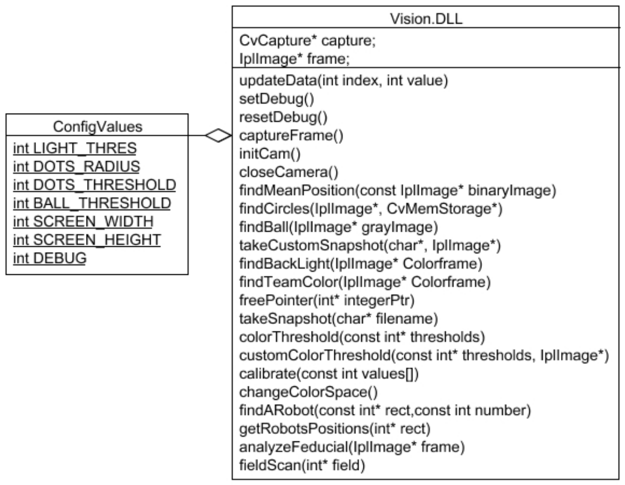
5.2.4 Implementation

The system was implemented using an Atmel's integrated development environment (IDE) called AVR Studio; this program simulates a environment similar to Microsoft's Visual Studio. We use AVR Studio to facilitate programming and debugging of our ATmega128 microcontroller.

5.3 Computer Vision System's Mid-Level System (MLS)

The main goal of the second software system, the MLS, is to provide multiple computer vision functions to allow the high-level system to locate and organize all the agents' information that is located within the area. Each function provides different information. For instance, a full *court scan* is used to find all agents; a partial *court scan* can look in a specific region of the court to find a specific agent or object. This system is written in C++ using the Intel's OpenCV). All the functions are grouped into a class, converted into a DLL, and then imported by the high-level system.

This MLS is also in charge of capturing all the pictures from the camera and processing the pictures. The MLS enables the users of the DLL to query the images captured so that live camera data can be provided for real-time video image processing. Since the system is sharing memory with other users, it must also provide a way to avoid memory leaks and manage all the memory that it allocates. Figure 9 is an overview of the main functions that the system will provide and the algorithms behind them.



```
Vision.DLL
CvCapture* capture;
IplImage* frame;
updateData(int index, int value)
setDebug()
resetDebug()
captureFrame()
initCam()
closeCamera()
findMeanPosition(const IplImage* binaryImage)
findCircles(IplImage*, CvMemStorage*)
findBall(IplImage* grayImage)
takeCustomSnapshot(char*, IplImage*)
findBackLight(IplImage* Colorframe)
findTeamColor(IplImage* Colorframe)
freePointer(int* integerPtr)
takeSnapshot(char* filename)
colorThreshold(const int* thresholds)
customColorThreshold(const int* thresholds, IplImage*)
calibrate(const int values[])
changeColorSpace()
findARobot(const int* rect,const int number)
getRobotsPositions(int* rect)
analyzeFeducial(IplImage* frame)
fieldScan(int* field)

ConfigValues
int LIGHT_THRES
int DOTS_RADIUS
int DOTS_THRESHOLD
int BALL_THRESHOLD
int SCREEN_WIDTH
int SCREEN_HEIGHT
int DEBUG
```

Figure 9 Vision Functions

The MLS interacts internally with two important pointers, CvCapture* and Ip1Image*. The CvCapture* pointer connects to the camera and all the frame capturing is done through this pointer. The IplImage* pointer determines the location in which the current frame is saved. Many of the functions that apply transformations to the frame will actually take this image, clone it, and perform the changes into a new IplImage* in such a way that the original captured frame remains unchanged. There are three functions that allow manipulation of these two objects:

- *initCam()* initializes the connection with the webcam and gets ready to capture.
- *captureFrame()* queries the webcam for a new frame and saves it into our frame pointer.
- *closeCamera()* closes the connection with the camera, thus freeing the resources.

The system has a set of configuration values that can be changed dynamically in order to calibrate the lighting and other values that are dependent on the lighting conditions that the camera perceives. The following three functions allow the user to change these values as necessary.

- *updateData*(int index, int value)
- *setDebug*()
- *resetDebug*()

There are many functions that perform transformations on the frame that have being stored in memory. The original frames are always untouched; these frames are cloned and then an operation is performed on the cloned frame.

Some of the most common functions are shown below.

- *findMeanPosition*(const IplImage* binaryImage) is meant to find the mean position of a set of white pixels in a binary image. This is an easy way to find the average location of contours.
- *findCircles*(IplImage*, CvMemStorage*) scans the whole image and finds circles in the image. A circle could potentially represent a ball, or any other round object.
- *findBall*(IplImage* grayImage) looks specifically for a ball.
- *takeCustomSnapshot*(char*, IplImage*) saves a picture of the given IplImage* object.
- *takeSnapshot*(char* filename) saves a picture of the current stored frame.
- *freePointer*(int* integerPtr) frees memory allocated by the DLL.
- *colorThreshold*(const int* thresholds) applies a filter to the saved frame image.
- *customColorThreshold*(const int* thresholds, IplImage*) applies a filter to the given IplImage*, and coverts all the pixels that do not fall within the ranges of the thresholds parameters to black.
- *changeColorSpace*() performs a real-time 8-bit conversion to the saved frame.

The algorithm to actually find the agents on the field is a little more involved that the functions mentioned above. The first requirement for this algorithm to function properly is that it the system has to be able to know determine the identity of each agent by identifying the uniform that each agent is wearing (see Figures 10 and 11). Each robot will wearing a sport T-shirt, with one of two colors (for the two teams). The color (either orange or blue) determines robot's team. Decorations in the form of dots are used to uniquely identify individual robots on a team. The area that the camera covers has a fairly uniform background of a yellow/green color (called green throughout the rest of this paper); this is intended to simulate the color of the grass on an outdoor field. Therefore, there will be three well identified colors on the field: orange, blue, and green. The first step in agent identification is to filter out all the green in an image. The *customColorThreshold* function is applied to transform all the green pixels and very white pixels into black. For all the pixels that are orange or blue this function writes (255,128,0) or (0,0,255), respectively in order to enforce the color for which we are searching. The resultant picture will feature various contours (blobs or outlines) of blue and orange. The next step is to identify the centroid for all the contours and to determine their sizes. After all the contours are located on the field, each blob is analyzed to determine the robot that is represented in that countor. The *AnalyzeFeducial* function will create a ROI (region of interest) around the contour, looking for a "backlight." The backlight is a rectangle that is located on the back part of the agent on the bottom of the shirt (see Figure 10). The centroid of this rectangle is then found. Given the centroid of this rectangle and the centroid of the shirt, a line can be drawn between the two to give a facing direction of the agent.

The next step is to analyze the color of the agent (blue or orange) to determine her team. After all the contours have been analyzed, an integer array (in the format [sizeOfArray, number of Robot, Color of shirt, CenterX of the shirt, CenterY of the Shirt, CenterX of the Backlight, CenterY of the Backlight, ...]) is returned. The size of the array is dynamically changed as more robots are introduced into the arena.

Recent experiments have shown that this algorithm, when applied to the entire arena, yields average results of 14 frames per second. When subsequent scans are required, a higher rate could be used since the previous position of each agent is known, and only small movement can occur in fractions of a second. Therefore, the program can look only in the neighborhood of the last know position of an agent to reacquire the agent's new position. Only when an agent has been lost entirely would a full scan be necessary. This algorithm generates all the information the high-level system needs in order to calculate the next move.

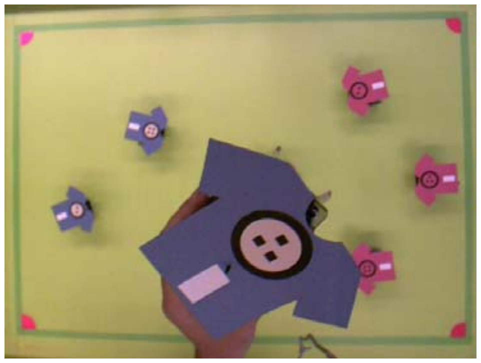


Figure 10 Field Camera Snapshot

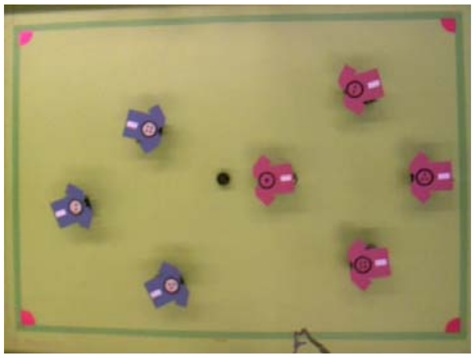


Figure 11 Field Camera Snapshot

5.4 High-Level System (HLS) and C# Implementation

The high-level system is the most important component of the entire project. The HLS integrates the other two systems and orchestrates their interactions. In addition, this system provides controls for the user to create new runtime behaviors for the agents. For example, the system offers a dynamic choreography editor in which the user can create a new plan that can be further edited and tested during runtime. The HLS uses its own compiler to run this choreography code. Since there will be eight agents running at the same time, this compiler runs code in a multi-threaded fashion to guarantee the parallel running of the code; this is crucial for the timing and for a successful execution of the choreography.

Another feature that the HLS hosts is the ability to create a dynamic speech creator, in which the user can choose from multiple voices and speeds, and can customize the text and presentation that the voices will articulate. The user can create different conversations, which can be dependent on agent behavior, or can sing songs.

Both the choreography and speech editors serve as user-friendly interfaces. The HLS also hosts the ability of creating task (or games) for the agents to perform. The ability to create new tasks dynamically without changing or compromising the current code for each agent is a definite advantage and will allow the system to evolve rapidly.

The HLS hosts a simulator and also integrates the two editors to communicate and analyze the camera feedback and send the result wirelessly to the agents. The main design goal for the HLS is to create a heavily object-oriented platform that can be easily expanded while providing robustness, modularity and functionality [9][10]. The HLS is divided into different Subsystems and their interactions as shown in Figure 12.

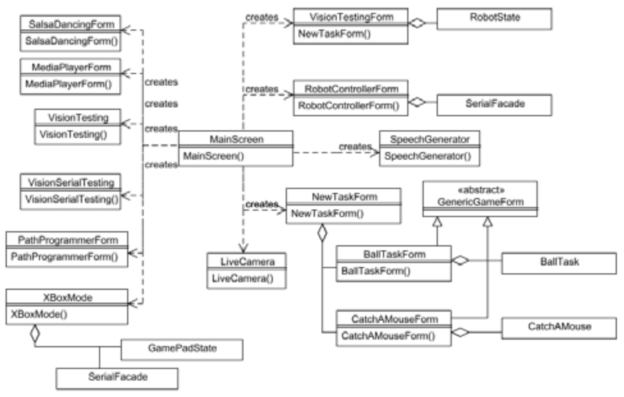


Figure 13 Subsystems Overview

The GUI subsystem (see Figure 13) integrates all the functions that the system supports and provides simple user interfaces to controls them. There are many different GUI's in the HLS, each provides different functionality. The HLS is meant to handle multiple windows at the same time. Each window is a child of the mdi parent window (which is the main window). The multimedia subsystem provides a simple interface to the user to incorporate sounds and videos that can be executed from the program using mci (media control interface), encapsulating all the details of usage and providing a simple interface for the user [9][11].

The *Speech Subsystem* uses the TTS (text to speech) interface to provide access to voices and speeches execution. The speech subsystem presents a unified Singleton/Facade that encapsulates all the interaction with the TTS and gives the user a simple and unified interface to be used with the system.

The *Task Subsystem* implements all the tasks that have been currently created. A task follows a well defined interface that provides all the necessary functionality to guarantee that the robots will try to complete a task and make it work.

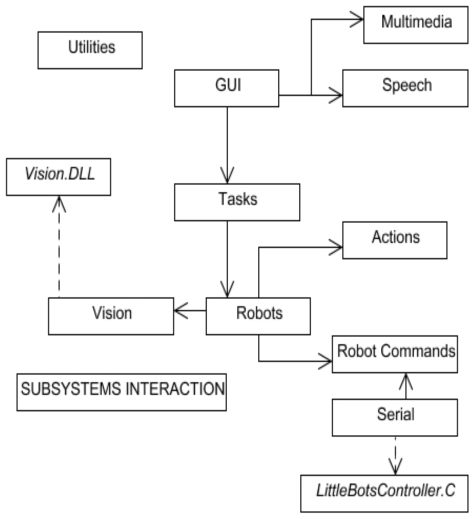


Figure 12 Graphical user interface Overview

The *Robot Subsystem* holds the code and behavior for different kind of agents. This subsystem provides different personalities to the robots so that they can act differently depending on the circumstances; it also hosts the code for the simulated agents.

The *Vision Subsystem* provides a well defined interface that encapsulates the interaction and synchronization of the Vision.DLL and provides the user with an interface that allows all the vision functionality without exposing implementation details.

The Actions Subsystem (see FIGURE 12) provides an interface to be implemented by the agents, each interface provides new functionality to the robots, to allow them to move around or to touch people or sense things, and depending on how complicated the robot wants to be.

The Robots Commands subsystem provides a hierarchy of commands that the robot performs and gets into a set format. These interfaces will know how to convert all the possible functions into the right commands that will be then sent into the serial subsystem.

Serial Subsystem This subsystem encapsulates all the process and information required by the system to communicate with the agents wirelessly. All the details are encapsulated and the subsystem provides a facade that internally manages and synchronizes all the serial requests. Finally the *Utilities* subsystem provides various function and classes that are used throughout the system.

6. ACKNOWLEDGMENTS

Sincerely appreciation is due to the Machine intelligence Laboratory at University of Florida and the National Science Foundation's Research Experience for Undergraduates (REU) program, and the for providing the means for this project to arise from ideas and discussion to actual realization. Special gratitude is due to our faculty advisors for all their support, encouragement, and advice. Finally, we are indebted to the many other members of the Machine Intelligence Laboratory (MIL) who have helped with advices and encouragement.

7. REFERENCES

[1] Christos Ampatzis, Elio Tuci, Vito Trianni, and Marco Dorigo. Evolution of Signalling in a Swarm of Robots Controlled by Dynamic Neural Networks. IRIDIA – *Technical Report Series No.TR/IRIDIA/2006-018, IRIDIA-Universit´e Libre de Bruxelles, Belgium*, September 2006.

[2] Sugawara, K.; Watanabe, T. Swarming robots - collective behavior of interacting robots**.** *Proceedings of the 41st SICE Annual Conference*, Volume 5, 5-7 Aug. 2002 Page(s):3214 - 3215 vol.5. IEEE SICE Conference Proceeding, 2002.

[3] Kevin Claycomb, Sean Cohen, Jacob Collums, Carlo Francis, Grzegorz Cieslewski, Tom Feeney, Donald J. Burnette, Gene Shokes, David Conrad, Hector Martinez III, Eric M. Schwartz. SubjuGator 2007. *Association for Unmanned Vehicle Systems International*, July 2007.

[4] Pack, D.J.; Mullins, B.E. Toward finding an universal search algorithm for swarm robots. *Intelligent Robots and Systems, 2003 Proceedings*. IEEE/RSJ International Conference on Volume 2, 27-31 Oct. 2003 Page(s):1945 - 1950 vol.2. IEEE Press. Las Vegas, Nevada, 2003.

[5] F. Mondada, M. Bonani, S. Magnenat, A. Guignard, and D. Floreano. Physical connections and cooperation in swarm robotics. In *Proc. of the* 8*th Conf. on Intelligent Autonomous Systems*, pages 53–60. IOS Press, Amsterdam, Netherlands, 2004.

[6] M. Rubenstein, K. Payne, P. Will, and W.-M. Shen. Docking among independent and autonomous CONRO self-reconfigurable robots. In *Proc. of the 2004 IEEE Int. Conf. on Robotics and Automation*, volume 3, pages 2877–2882. IEEE Computer Society Press, Los Alamitos, CA, 2004

[7] Marco Dorigo. Swarm-bot: An Experiment in Swarm Robotics. *Proceedings of SIS 2005 - IEEE Swarm Intelligence Symposium*, IEEE Press, 241–248.

[8] Dongtang Ma; Jibo Wei; Zhaowen Zhuang. A novel optical signal detecting and processing method for swarm robot vision system. *Robotics, Intelligent Systems and Signal Processing, 2003. Proceedings*. IEEE International Conference on Volume 2, 8-13 Oct. 2003 Page(s):1081 - 1085 vol.2. IEEE press. Changsha, China, October 2003.

[9] Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, Morgan Skinner. *Professional C# 2005 with .NET 3.0Professional Guide).* Wiley Publishing, Indianapolis, IN, 2007.

[10] Meilir Page-Jones. *Fundamentals of Object-Oriented Design in UML*. Addison-Wesley, Reading, MA, 1995.

[11] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.